Encapsulation

**Information Hiding**

- Programmer using a class method need **not** know details of implementation
  - Only needs to know *what* the method does

- **Information hiding:**
  - Designing a method so it can be used without knowing details

- Also referred to as *abstraction*

- Method design should separate *what* from *how*

**Encapsulation**

- **Encapsulation:** Hiding implementation details of an object from its clients.
  - Encapsulation provides *abstraction*.
    - separates external view (behavior) from internal view (state)
    - Encapsulation protects the integrity of an object’s data.
When Creating Classes

- When creating the public interface of a class, give careful thought and consideration to the contract you are creating between yourself and users (other programmers) of your class.

- Use *preconditions* to state what you assume to be true before a method is called.
  - caller of the method is responsible for making sure these are true.

- Use *postconditions* to state what you guarantee to be true after the method is done if the preconditions are met.
  - implementer of the method is responsible for making sure these are true.

Pre- and Postcondition Comments

- **Precondition comment**
  - States conditions that must be true before method is invoked

- **Example**

```java
/**
 * Precondition: The instance variables of the calling object have values.
 * Postcondition: The data stored in (the instance variables of) the receiving object have been written to the screen.
 */
public void writeOutput()
```

Visibility Modifiers

- All parts of a class have *visibility modifiers*.
  - Java keywords: `public`, `protected`, `private`.
  - Do not use these modifiers on local variables (syntax error).

- **public** means that constructor, method, or field may be accessed outside of the class.
  - part of the interface
  - constructors and methods are generally public.

- **private** means that part of the class is hidden and inaccessible by code outside of the class.
  - part of the implementation
  - data fields are generally private.
The **public** and **private** Modifiers

- Type specified as **public**
  - Any other class can directly access that object by name

- Classes are generally specified as **public**

- Instance variables are usually **not public**
  - Instead specify as **private**

Accessing private state

- We can provide methods to get and/or set a field’s value:
  ```java
  // A "read-only" access to the x field ("accessor")
  public int getX() {
    return x;
  }

  // Allows clients to change the x field ("mutator")
  public void setX(int newX) {
    x = newX;
  }
  ```

- Client code will look more like this:
  ```java
  System.out.println("p1: (" + p1.getX() + ", " + p1.getY() + ")");
  p1.setX(14);
  ```

Private fields

- A field can be declared **private**.
  - No code outside the class can access or change it.
    ```java
    private type name;
    ```

- Examples:
  ```java
  private int id;
  private String name;
  ```

- Client code sees an error when accessing private fields:
  ```java
  PointMain.java:11: x has private access in Point
  System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
  ^
  ```

Programming Example

```java
public class Rectangle {
  private int width;
  private int height;
  private int area;

  public void setDimensions (int newWidth, int newHeight) {
    width = newWidth;
    height = newHeight;
    area = width * height;
  }

  public int getArea () {
    return area;
  }
}
```

- Statement such as `box.width = 6;` is **illegal** since `width` is **private**
- Keeps remaining elements of the class consistent

Note the `setDimensions` method:
- This is the only way the `width` and `height` may be altered outside the class
// A Point object represents an (x, y) location.
public class Point {
    private int x;
    private int y;
    public Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }
    public double distanceFromOrigin() {
        return Math.sqrt(x * x + y * y);
    }
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
    public void setLocation(int newX, int newY) {
        x = newX;
        y = newY;
    }
    public void translate(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }
}

Client code

public class PointMain4 {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point(5, 2);
        Point p2 = new Point(4, 3);
        // print each point
        System.out.println("p1: (" + p1.getX() + ", " + p1.getY() + ")");
        System.out.println("p2: (" + p2.getX() + ", " + p2.getY() + ")");
        // move p2 and then print it again
        p2.translate(2, 4);
        System.out.println("p2: (" + p2.getX() + ", " + p2.getY() + ")");
    }
}

OUTPUT:
p1 is (5, 2)
p2 is (4, 3)
p2 is (6, 7)

Encapsulation

- Consider example of driving a car
  - We see and use break pedal, accelerator pedal, steering wheel
    - know what they do
  - We do not see mechanical details of how they do their jobs

- Encapsulation divides class definition into
  - Class interface
  - Class implementation

Encapsulation

- Class interface
  - Tells what the class does
  - Gives headings for public methods and comments about them

- Class implementation
  - Contains private variables
  - Includes definitions of public and private methods
Encapsulation

- A well encapsulated class definition

**Class Definition**

- Implementation:
  - Private instance variables
  - Private constants
  - Bodies of public methods

- Interface:
  - Comments
  - Headings of public methods
  - Public named constants

**Programmer who uses the class**

Benefits of encapsulation

- Provides abstraction between an object and its clients.
- Protects an object from unwanted access by clients.
  - A bank app forbids a client to change an Account’s balance.
- Allows you to change the class implementation.
  - Point could be rewritten to use polar coordinates (radius $r$, angle $\theta$), but with the same methods.
- Allows you to constrain objects’ state (invariants).
  - Example: Only allow Points with non-negative coordinates.

Encapsulation – Best Practices

- Preface class definition with comment on how to use class.
- Declare all instance variables in the class as private.
- Provide public accessor methods to retrieve data and provide public methods to manipulate data
  - Such methods could include public mutator methods.
- Place a comment before each public method heading that fully specifies how to use method.
- Make any helping methods private.
- Write comments within class definition to describe implementation details.

Software Development Observations

- Interfaces change less frequently than implementations.
- When an implementation changes, implementation-dependent code must change accordingly.
- Hiding the implementation reduces the possibility that other program parts will become dependent on class-implementation details.
// Fig. 8.1: Time1.java
public class Time1
{
    private int hour; // 0 – 23
    private int minute; // 0 – 59
    private int second; // 0 – 59

    public void setTime(int h, int m, int s)
    {
        hour = (h >= 0 && h < 24) ? h : 0; // validate hour
        minute = (m >= 0 && m < 60) ? m : 0; // validate minute
        second = (s >= 0 && s < 60) ? s : 0; // validate second
    }

    public String toUniversalString()
    {
        return String.format("%02d:%02d:%02d", hour, minute, second);
    }

    public String toString()
    {
        int hour1 = (hour == 0 || hour == 12) ? 12 : hour % 12;
        return String.format("%d:%02d:%02d %s", hour1, minute, second, (hour < 12 ? "AM" : "PM");
    }
}

/*
 * Time1Test.java
 */

public class Time1Test
{
    public static void main(String args[])
    {
        // create and initialize a Time1 object
        Time1 time = new Time1();
        // invokes Time1 constructor

        // output string representations of the time
        System.out.print("The initial universal time is: ");
        System.out.println(time.toUniversalString());
        System.out.print("The initial standard time is: ");
        System.out.println(time.toString());
        System.out.println();

        // change time and output updated time
        time.setTime(13, 27, 6);
        System.out.print("Universal time after setTime is: ");
        System.out.println(time.toUniversalString());
        System.out.print("Standard time after setTime is: ");
        System.out.println(time.toString());
        System.out.println();

        // set time with invalid values; output updated time
        time.setTime(99, 99, 99);
        System.out.println("After attempting invalid settings:");
        System.out.print("Universal time: ");
        System.out.println(time.toUniversalString());
        System.out.print("Standard time: ");
        System.out.println(time.toString());
    }
}
Performance Tip

- Java conserves storage by maintaining only one copy of each method per class
  - this method is invoked by every object of the class.
- Each object, on the other hand, has its own copy of the class’s instance variables (i.e., non-static fields).
- Each method of the class implicitly uses this to determine the specific object of the class to manipulate.

Software Development Observations & Tips

- When one object of a class has a reference to another object of the same class, the first object can access all the second object’s data and methods (including those that are private).
- When implementing a method of a class, use the class’s set and get methods to access the class’s private data. This simplifies code maintenance and reduces the likelihood of errors.
- This architecture helps hide the implementation of a class from its clients, which improves program modifiability

Default and No-Argument Constructors

- Every class must have at least one constructor
  - If no constructors are declared, the compiler will create a default constructor
    - Takes no arguments and initializes instance variables to their initial values specified in their declaration or to their default values
      - Default values are zero for primitive numeric types, false for boolean values and null for references
    - If constructors are declared, the default initialization for objects of the class will be performed by a no-argument constructor (if one is declared)

Common Programming Error

- If a class has constructors, but none of the public constructors are no-argument constructors, and a program attempts to call a no-argument constructor to initialize an object of the class, a compilation error occurs.
  - A constructor can be called with no arguments only if the class does not have any constructors (in which case the default constructor is called) or if the class has a public no-argument constructor.
**final Instance Variables**

- **final instance variables**
  - Keyword **final**
    - Specifies that a variable is not modifiable (is a constant)
    - **final** instance variables can be initialized at their declaration
      - If they are not initialized in their declarations, they must be initialized in all constructors

- If an instance variable should not be modified, declare it to be **final** to prevent any erroneous modification.

**static final Instance Variables**

- A **final** field should also be declared **static** if it is initialized in its declaration.
- Once a **final** field is initialized in its declaration, its value can never change.
- Therefore, it is not necessary to have a separate copy of the field for every object of the class.
- Making the field **static** enables all objects of the class to share the **final** field.
- Example: `public static final double PI = 3.141592;`

**Software Reusability**

- Rapid application development
  - Reusability speeds the development of powerful, high-quality software

- Java’s API
  - provides an entire framework in which Java developers can work to achieve true reusability and rapid application development
  - Documentation:
    - [java.sun.com/j2se/5.0/docs/api/index.html](http://java.sun.com/j2se/5.0/docs/api/index.html)
    - Or [java.sun.com/j2se/5.0/download.html](http://java.sun.com/j2se/5.0/download.html) to download

- **Good Programming Practice:** Avoid reinventing the wheel. Study the capabilities of the Java API. If the API contains a class that meets your program’s requirements, use that class rather than create your own.

**UML Class Diagrams**

- An automobile class outline as a UML class diagram

```
+ Automobile
  - fuel: double
  - speed: double
  - license: String
  + accelerate(double pedalPressure): void
  + decelerate(double pedalPressure): void
```
### UML Class Diagrams

- Example: **Purchase**
  - Class

  ```
  Purchase
  - name: String
  - itemCount: int
  - itemPrice: double
  - numberOfBought: int
  
  + setName(String newName): void
  + setPrice(int count, double costPerCount): void
  + setNumberOfBought(int number): void
  + readInput(): void
  + writeOutput(): void
  + getName(): String
  + getTotalCost(): double
  + getUnitCost(): double
  + getNumberOfBought(): int
  ```

  - Plus signs imply public access
  - Minus signs imply private access

### Packages and Importing

- A **package** is a collection of classes grouped together into a folder
- Name of folder is name of package
- Each class
  - Placed in a separate file
  - Has this line at the beginning of the file
  ```
  package Package_Name;
  ```
- Classes use packages by use of **import** statement

### UML Class Diagrams

- Contains more than interface, less than full implementation
- Usually written *before* class is defined
- Used by the programmer defining the class
  - Contrast with the *interface* used by programmer who uses the class

### Package Names and Directories

- Package name tells compiler path name for directory containing classes of package
- Search for package begins in class path base directory
  - Package name uses dots in place of / or \n- Name of package uses relative path name starting from any directory in class path
Package Names and Directories

- A package name

Time Class Case Study: Creating Packages

- To declare a reusable class
  - Declare a public class
  - Add a package declaration to the source-code file
    - must be the very first executable statement in the file
    - Package name example: com.deitel.jhtp6.ch08
    - package name is part of the fully qualified class name
      » Distinguishes between multiple classes with the same name belonging to different packages
      » Prevents name conflict (also called name collision)

Example

- Time1.java

```java
public class Time1 {
  private int hour; // 0 - 23
  private int minute; // 0 - 59
  private int second; // 0 - 59

  public void setTime(int h, int m, int s) {
    hour = ((h >= 0 && h < 24) ? h : 0); // validate hour
    minute = ((m >= 0 && m < 60) ? m : 0); // validate minute
    second = ((s >= 0 && s < 60) ? s : 0); // validate second
  }
}
```

Time Class Case Study: Creating Packages (Cont.)

- Compile the class so that it is placed in the appropriate package directory structure
  - Example: our package should be in the directory
    - `com.deitel.jhtp6.ch08`

- javac command-line option `-d`
  - `javac` creates appropriate directories based on the class’s package declaration
  - A period (.) after `-d` represents the current directory
Time Class Case Study: Creating Packages (Cont.)

- Import the reusable class into a program
  - Single-type-import declaration
    - Imports a single class
    - Example: import java.util.Random;

- Type-import-on-demand declaration
  - Imports all classes in a package
  - Example: import java.util.*;

Name Clashes

- Packages help in dealing with name clashes
  - When two classes have same name

- Different programmers may give same name to two classes
  - Ambiguity resolved by using the package name

Overloading Basics

- When two or more methods have same name within the same class

- Java distinguishes the methods by number and types of parameters
  - If it cannot match a call with a definition, it attempts to do type conversions

- A method’s name and number and type of parameters is called the signature

Programming Example

```java
/** This class illustrates overloading. */
public class Overload {
    public static void main (String [] args) {
        double average1 = Overload.getAverage (40.0, 50.0);
        double average2 = Overload.getAverage (1.0, 2.0, 3.0);
        char average3 = Overload.getAverage ('a', 'c');
        System.out.println ("average1 = " + average1);
        System.out.println ("average2 = " + average2);
        System.out.println ("average3 = " + average3);
    }
    public static double getAverage (double first, double second) {
        return (first + second) / 2.0;
    }
    public static double getAverage (double first, double second, double third) {
        return (first + second + third) / 3.0;
    }
    public static char getAverage (char first, char second) {
        return (char) ((((int) first + (int) second)) / 2);
    }
}
```

```plaintext
average1= 45.0
average2= 2.0
average3 = b
```
Overloading and Type Conversion

- Overloading and automatic type conversion can conflict
- Remember the compiler attempts to overload before it does type conversion
- Use descriptive method names, avoid overloading when possible

Overloading and Return Type

- You must not overload a method where the only difference is the type of value returned

```
/**
 * Returns the weight of the pet.
 */
public double getWeight()

/**
 * Returns '=' if overweight, '>' if underweight, and '=' if weight is OK.
 */
public char getWeight()
```

Summary

- Precondition comment states conditions that must be true before method invoked
- Postcondition comment describes resulting effects of method execution
- Usage of visibility modifiers for encapsulation
- Separation of interface and implementation is important
- Class designers use UML notation to describe classes
- Use packages for software reusability
- Overloading must be done with care

Acknowledgments

- The course material used to prepare this presentation is mostly taken/adopted from the list below:
  - Java - An Introduction to Problem Solving and Programming, Walter Savitch, Pearson, 2012
  - Java - How to Program, Paul Deitel and Harvey Deitel, Prentice Hall, 2012
  - Mike Scott, CS314 Course notes, University of Texas Austin